Beating the Stock Market with Physics Informed Neural Networks: Comparing Jump-Diffusion Simulation to Deep Learning

Brigham Freeman
(Dated: November 4, 2025)

ABSTRACT

Physics-Informed Neural Networks (PINNs) are an emerging machine learning framework that embed mathematical constraints within the learning process to ensure consistency with physical or quantitative laws. Building upon Robert C. Merton's Jump-Diffusion Model, which captures the random fluctuations in stock prices, this paper develops a PINN capable of learning underlying market dynamics and simulating stock price behavior.

To evaluate the performance of this physics-based approach, a CNN-LSTM hybrid model was constructed as a benchmark for stock price forecasting. For six-month prediction windows, the CNN-LSTM achieved mean absolute error (MAE) rates of 4.80% on training data and 2.78% on unseen data, while the PINN-based simulations achieved 21.2% and 12.64%, respectively. Although the CNN-LSTM outperformed the PINN in direct price forecasting, the PINN produced accurate Jump-Diffusion simulations within an average MAE of $6.45 and successfully learned interpretable market parameters that describe price behavior.

The results indicate that CNN-LSTM models are more effective for direct price prediction, whereas PINNs are better suited for extracting market parameters and understanding the mechanisms driving stock price movements. An ensemble of both approaches may provide the most comprehensive framework for stock market modeling and analysis.

INTRODUCTION

One of the most famous examples of overlap between the disciplines of physics and finance is the Black-Scholes equation. The Black-Scholes equation is a differential equation that models European-style options trading. Quantitative finance analysts make use of the Black-Scholes equation to model stock market behavior [1].

While the Black-Scholes equation is useful, an expansion to it is better for modeling volatile stocks. The Merton Jump-Diffusion model is an expansion of the Black-Scholes equation that adds a random "jump" term. This jump models the fluctuations in stock prices that occur due to earnings announcements, market crashes, economic policy, or other factors that fall outside the market itself. As a result, the Jump-Diffusion model can be used to model extreme events that traditional forecasting methods may miss [1].

The Merton Jump-Diffusion model is given as [1][2]

$$dS_t = \mu S_t dt + \sigma S_t dW_t + S_t dJ_t \tag{1}$$

While not a traditional machine learning model, the Merton-Jump Diffusion model can be used to simulate market behavior and solve for different market parameters [2]. If the parameters are properly tuned with optimization, the model can produce data close to the true market behavior. Because random variance is inherent to the Jump-Diffusion model, predictions tend to drift a bit from small fluctuations in true data, but they can fit trends in data very well. An example of this is given in Figure 1, in which a stock price was simulated using Scipy optimization libraries to tune Merton-Jump Diffusion parameters to the stock data.

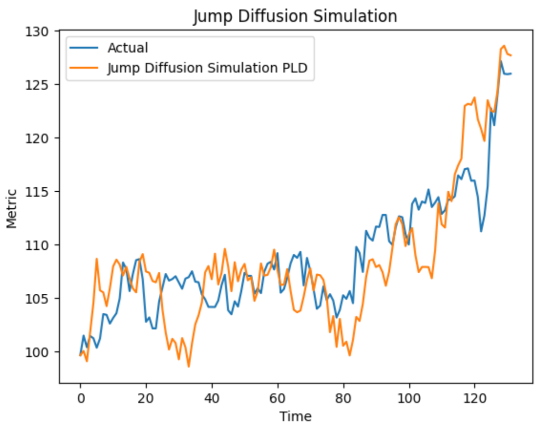

FIG. 1. Jump-Diffusion Modeling of Stock Price

Due to the dynamic behavior of the stock market, traditional machine learning models, like LSTM for time series, are often unable to capture market dynamics. This is because machine learning models rely on continuously differentiable functions for gradient descent. While it can be done, it is challenging to apply machine learning models to volatile stock data without significant smoothing, which can reduce the effectiveness of the model on real data.

Physics-Informed Neural Network

Physics Informed Neural Networks (PINN) are an emerging deep learning framework to combine mathematical models and data [3]. A hallmark feature of PINN is the respect to given laws of physics in their learning. Instead of using a physics law to govern learning, a PINN could instead be trained on other mathematical or physics formulas.

The goal of this project was to build a PINN that utilizes the Jump-Diffusion model to learn the Jump-Diffusion parameters, as opposed to forecasting strict price data. By learning how to use the Jump-Diffusion equation, the model learns how to craft a model of the stock market that accurately captures volatile behavior. It is also less time-intensive to forecast future state with the Jump-Diffusion model compared to a neural network, allowing for improved short-term forecasting.

The model evaluates the residual of the Jump-Diffusion equation, as well as the difference between the learned and observed parameters and values. Over time, this supports convergence of learning onto an optimal Jump-Diffusion equation with tuned parameters to effectively model a wide range of individual stocks, or entire market segments.

MODEL SETUP

The data used to train the model is readily available from Yahoo Finance. Using the yfinance library, stock data from the past six months is imported into a Colab Notebook. This provides daily updates to the model to help it stay relevant with a changing market. All of the data is then turned into a Pandas DataFrame for easy data manipulation. Two sets were created from the main DataFrame, each containing the stock tickers for the training and validation data. The first set is the training data, which the model uses to learn parameters. The second set is used after learning to determine model performance on unseen data. This benchmark is used to gauge the effectiveness of the learned parameters.

To learn how to build the Jump-Diffusion model, two neural networks are used. They are $\hat{V}$ and a separate network to track parameter values. The model $\hat{V}$ is a fully connected dense model with seven layers. $\hat{V}$ is used to learn the residual of the Jump-Diffusion model. The layers have the following setup:

Layer	Neurons	Initialization
1	128	tanh
2	96	tanh
3	64	tanh
4	48	tanh
5	32	tanh
6	16	tanh
7	1	None

TABLE I. Main $\hat{V}$ Network

The parameters for the jump diffusion model are tracked by a second neural network. In the `__init__()` function, the variables are initialized as weights with no shape. They are named after the parameters they represent. The initial weights are given in Table II.

The loss function is given as

$$\text{loss} = \alpha(\hat{V}_{sim} - V_{obs}) + V_{\text{Residual}} \tag{2}$$

In this equation, $\hat{V}_{sim}$ is created using the output of the parameter neural network. The found parameters are used to create a Jump-Diffusion simulation of the stock data on hand. V_{obs} is the function to generate observed data, given by

Weight	Initial Value
μ	0.002
σ	0.2
λ	0.02
β	0.0
γ	0.1

TABLE II. Initial Parameter Weights

$$V_{obs}(S) = S + S\mu dt + \sigma S dW + S\lambda(\beta + \frac{1}{2}\gamma) \tag{3}$$

where β is the parameter for mean jump size and γ is the standard deviation of the jump. While V_{obs} doesn't include the random discrete jumps of the usual Jump-Diffusion model, it uses the learned λ and γ to create stochastic smooth jumps. By using smooth functions, the model is able to use gradient descent to learn parameters. V_{obs} is created by using Monte Carlo averaging with current parameter values to reduce noise. This helps improve convergence and leads to more stable parameter updates.

A total of 4000 epochs are run. A cosine decay learning rate is used, with an initial learning rate of 0.0001 and 4000 decay steps. This is to help the model converge better later in training. In the loss function, α was set to 0.5 to lower the impact of observed data. V_{obs} was created by running a jump-diffusion simulation using the current parameter values across 20 iterations. In training, the data is scaled to the range $(0, 1)$. This helps identify general trends across all stocks and eliminate the impact of outliers, especially for the μ parameter. Since μ is functionally similar to the slope, a stock with prices centered around \$600-\$650 would skew the parameter much higher than stocks in the \$10-\$15 range.

To prevent the model from getting trapped in local minima, random variable mutations were introduced. Mutations occur at 200, 400, 800, 1600, and 3200 epochs, as well as whenever there is no improvement in loss for 300 consecutive epochs. The model maintains a record of the best solution found to date, allowing it to preserve optimal parameters if a mutation temporarily degrades performance. This is achieved through two variables, `best_weights_vhat` and `best_weights_param`, which are initialized to `None` at epoch 0. The following logic is used to track and store the best weights:

```
if loss.numpy() < best_loss - 1e-5:
    best_loss = loss.numpy()
    best_weights_v = v_hat.get_weights()
    best_weights_p = params.get_weights()
```

Since these variables store the weights of `param_net`, calling `get_weights()` saves the best parameters for the system. At the end of training, the networks' weights are reset to these stored values.

This mutation mechanism is inspired by genetic algorithm optimization, where random perturbations are used to explore new regions of the loss landscape. By restoring the best weights after each mutation cycle, the model can explore alternative solutions without permanently settling in suboptimal minima.

The model was evaluated by constructing Jump-Diffusion simulations using the found parameter values. The loss values for testing were calculated using the difference between simulated stock prices and true stock prices. This tested the ability of the found parameters to construct stock data that fits overall market trends.

Model Results

The PINN model performed relatively well, especially when the randomness of the Jump-Diffusion model is taken into account. On training data, the simulations produced an average mean squared error of 9,787.81. While the MSE is large, it produced a mean absolute error of only 6.45. The error measurements for unseen data were 9,452.50 and 13.70 respectively.

To reduce the impact that high-value stocks have on the MAE, a MAE % was introduced. This would track percent differences between actual and observed data, a more fair metric across all stocks. The PINN produced an average mean absolute error rate of 21% on training data, and 12% on unseen data. The full values are included in the table below.

Run	MSE (Seen Data)	MAE (Seen Data)	MAE % (Seen Data)
1	6,657.19	10.27	22%
2	5,766.56	5.36	24%
3	14,537.72	20.62	20%
4	4,783.58	4.27	18%
5	17,194.00	17.61	22%
AVG	9,787.81	6.45	21.2%

TABLE III. PINN Training Data Measurements With Averages

Once the model was exposed to the training data, it was then given a set of 47 unseen stocks. The results of the runs on the unseen data are given below.

Run	MSE (New Data)	MAE (New Data)	MAE % (New Data)
1	13396.41	13.20	14%
2	10649.37	19.22	14%
3	7755.37	8.69	13%
4	8267.58	29.06	9%
5	7193.78	32.72	13%
AVG	9452.50	13.70	12.64%

TABLE IV. PINN Training Data Measurements With Averages for Unseen Data

The average mean absolute error percent was 8.56% lower on unseen data, while the mean absolute average error was almost double that of the seen data. This suggests potential overfitting on training data. Since the training data set is much larger than the unseen test data set, it's also possible that there were more volatile stocks in the training set than in the test set. The predictions also modeled the volatility inherent to stock prices, matching data other models struggle to.

Comparison to CNN-LSTM Hybrid Network

To compare the data from the PINN to other neural networks, a CNN-LSTM hybrid model was constructed. This framework was chosen due to CNN and LSTM being some of the most common time series forecasting options, providing a baseline forecasting method to compare the PINN to [2]. Unlike the PINN, the LSTM model simply predicts stock prices, not overarching market parameters. There were several notable differences between the Jump-Diffusion simulations and the predictions from the LSTM model. The setup for the CNN-LSTM model is given in Table V.

Layer	Type	Neurons/Parameters	Initialization
1	Conv1D	`iters=13, kernel_size = 3`	tanh
2	BatchNormalization		
3	MaxPooling1D	`pool_size=2`	
4	LSTM	64	tanh
5	LSTM	32	tanh
6	Dense	1	None

TABLE V. CNN-LSTM Hybrid Network

The hybrid network used a standard 80%/20% training/validation data split. The model was defined and compiled prior to training, with 25 epochs being fit per stock data. This was done so the model learns across stock data, and doesn't reset training between every stock. Prior to being used for training, data was scaled to the range $(-1, 1)$ with a MinMaxScaler from Scikit-learn. This range was chosen to fit data to the tanh activation function. To prevent data leakage from the scaler, the test and validation sets were split prior to being scaled independently. Similar to the PINN, the goal was to train a general model across all sectors of the stock market.

After training across all regular stock data, predictions were fit for the unseen stocks. This was done using TensorFlow's `.predict()` method without fitting the new data. Other hyperparameter settings was an input of 10 time steps, and a batch size of 32 for training. The results for the CNN-LSTM are given below.

Run	MSE	MAE	MAE %
Seen Data Averages	384.01	7.35	4.91%
Unseen Data Averages	168.90	4.24	2.46%

TABLE VI. PINN Training Data Measurement Averages for CNN-LSTM

The CNN-LSTM model had surprisingly low loss values, outperforming the PINN by 16.29% on seen and 10.18% on unseen data. A hypothesis initially formed during research was that the PINN would outperform the CNN-LSTM due to volatile data, which was proven to be untrue. However, this may have been an issue with the data structure. The CNN-LSTM was trained on the same six months of stock data, meaning that only about 36 days of data was used to formulate predictions. This likely kept error measurements low, since there was a smaller time frame for predictions to drift.

To test this, a new DataFrame was constructed that extended price history to 24 months. The `train_size` parameter was also lowered to 0.75, which lengthened the prediction time frame from 36 days to a full six months.

Run	MSE	MAE	MAE %
Seen Data Averages	368.37	7.18	4.80%
Unseen Data Averages	175.41	4.63	2.78%

TABLE VII. PINN Training Data Measurement Averages for CNN-LSTM

The hybrid CNN-LSTM model performed much better than expected. The convolutional layers help the model detect local patterns and short-term fluctuations in the data, while the LSTM layers capture longer-term temporal dependencies. By combining both, the model can learn both short- and long-term trends, resulting in more accurate predictions. A graph of predictions vs. actual results is given below.

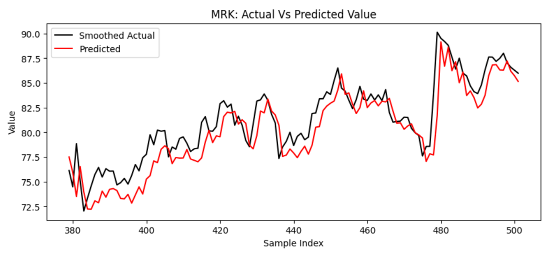

FIG. 2. Predictions vs. actual stock prices.

RESULTS

The hybrid CNN-LSTM model performed strongly on stock data. Although initially expected to under perform the PINN due to gradient descent issues, it proved highly effective once the data was scaled. Even with the six month prediction window, the CNN-LSTM outperformed the PINN on seen data by 16.4 %, and by 9.85% on unseen data. Despite its shallow depth, the CNN-LSTM hybrid is a valid, and often preferable, approach for stock price forecasting.

While the PINN is designed to estimate Jump-Diffusion parameters rather than prices directly, the results can be used to easily construct forecasts from the Jump-Diffusion model. The CNN-LSTM is much better at predicting prices than these models, although it does not capture the parameters that describe why the stock/market behaves as it does.

There is an inherent disadvantage to modeling stock prices via the Jump-Diffusion model, namely, the random behavior. This skews the results of the simulations built from the PINN, causing poorer performance. Despite this random variance, the PINN performed relatively well, often producing similar plots to the CNN-LSTM. The major tradeoff between the two models has less to do with raw stock price prediction, and more to do with learning parameters.

In scenarios where predicting the raw stock price is the primary goal, the CNN-LSTM model may be more effective. However, when the objective is to learn underlying market parameters, such as drift, volatility, or jump behavior, the PINN approach offers significant advantages. This contrast highlights a fundamental difference between traditional forecasting and physics-informed models: PINNs aim to capture the *mechanisms* driving the data, not just the outcomes. By understanding the system governing stock price dynamics, analysts gain deeper insights to make informed decisions, rather than relying solely on predicted prices. Due to the differences in what each network is attempting to do, the best approach for analysts to predict prices and uncover trends across the market is to use the two in tandem. A Colab notebook with this approach is linked in the Appendix.

Accuracy Notice

Because this project is tied to current market values, the performance and loss values *will* change dependent on market conditions. The majority of the testing was done between October 30th - November 2nd, 2025. Due to fluctuating market trends, the loss values will change on a regular basis. Based on further testing, the drift appears to be +/- 1.5 for MAE %.

[1] Merton, R. C. (1976). Option pricing when underlying stock returns are discontinuous. Journal of Financial Economics, 3(1–2), 125–144. https://doi.org/10.1016/0304-405X(76)90022-2

[2] Kou, Steven, A Jump Diffusion Model for Option Pricing (August 2001). https://ssrn.com/abstract=242367 or http://dx.doi.org/10.2139/ssrn.242367

[3] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, Journal of Computational Physics, Volume 378, 2019, Pages 686-707, ISSN 0021-9991, https://doi.org/10.1016/j.jcp.2018.10.045.

[4] Kong, X., Chen, Z., Liu, W. et al. Deep learning for time series forecasting: a survey. Int. J. Mach. Learn. & Cyber. 16, 5079–5112 (2025). https://doi.org/10.1007/s13042-025-02560-w

APPENDIX

Colab Code: Linked Here